A Measurement-Based Framework for Software Reliability Improvement

Karama Kanoun

A Measurement-Based Framework
for Software Reliability Improvement

Karama Kanoun

LAAS-CNRS — 7 Avenue du Colonel Roche
31077 Toulouse Cedex 4 — France
kanoun@laas.fr

Abstract

Programs for software reliability improvement based on measurements require the collection and analysis of comprehensive and consistent data sets on several software projects. In this paper, we put emphasis on data collection and analysis programs for software reliability improvement. We first present the objectives of data collection programs, report some success stories related to software reliability improvement, then discuss the practical aspects of data collection, validation and processing before giving recommendations for successful data collection and analysis programs. The success stories show that the gain in productivity and reliability is obtained at almost no extra cost, and even with an overall cost reduction most of the time. Data processing consists in performing statistical treatments. For reliability purposes, we consider three main activities: descriptive analysis, trend analysis, and reliability evaluation. The recommendations and examples of results, given at the end of the paper, are based on our experience in processing failure data collected on real-life software systems. We discuss in particular, the relevance of software reliability evaluation according to the life-cycle phase considered.

1. Introduction

The need for improving system dependability has risen dramatically over the past few decades. Achieving dependability requires painstaking efforts and a high level of commitment of all those involved in the design and production, from specifications to operation and maintenance. A fully integrated approach, based on qualitative and quantitative aspects, is needed to ensure that dependability is taken into account as early as the specification phase and correctly handled, and the expected goals are reached for the final product.

From a practical viewpoint, very early and more recent published work show the continuous growth of software cost in the total system's cost (see e.g., [Boehm 1981; Castelli *et al.* 1997]). Also, the increasing role of the software in computer-based systems is confirmed when considering the sources of failures reported from the field. The latter show that the software is responsible for a large amount of system failures. For example, a study by Harris Research found that every American worker who uses a personal computer loses up to three weeks a year to system problems and a British study calculated that software-related crashes were costing the country two million worker-days each year. Hence the need for improving software reliability.

Measurement is the first step towards software reliability improvement and system outage reduction. It is essential for understanding the underlying phenomena, making correlation between observations, identifying weakness, controlling important issues and improving the development process and the product. Indeed, measurement is an important part of the software engineering activities.

Several attributes can be measured, relating to:

- The product itself: its size, complexity, modularity, re-use, control flow and data flow.

1

- The development process, including: productivity, development team communication, cost, effort and schedule.

- Deployment and operation: installation, maintenance and upgrading, user and programmer documentation.

- Quality and reliability: failures, corrections, fault density, time between failures and other measures.

All of these are valuable and are used in a way or another to improve the software or the development and maintenance process. However, when initiating a measurement process, it is worthwhile to establish specific objectives: begin with a small number of attributes focused on a limited set of objectives. Later, after the first success, the scope can be enlarged and more attributes can be addressed. An incremental process allows progressive mastering of the difficulties encountered.

Indeed, due to the prominent role of data collection in producing high-quality software, practically all companies involved in developing large-scale software projects have initiated a data collection process. Reliability improvement programs are one tool used by companies to ameliorate the maturity of their software production process, within the context of broader process improvement efforts such as that used in Japan [Aoyama 1993], or the Capability Maturity Model (CMM) originating from the SEI [Humphrey 1989].

In this paper, our goal is to motivate those involved in software development, who have some experience but had never plucked up their courage to start collecting reliability data in a systematic way. Also, we aim at encouraging those who are aware of the need for measurement but are not confident that they can do efficiently or effectively. Our objective is to give them an overview and examples of data to be collected and how to process them to obtain quick feedback.

2

A large part of our recommendation results from the interactions we had with small and larger companies, with whom we worked to process failure data or to establish a data collection process. Based on our experience, we have defined a framework allowing rigorous processing of collected data for software reliability analysis and evaluation. In [Kanoun *et al.* 1997], we put emphasis on failure data processing and presented an application to a real-life system whereas in this paper we place emphasis on the practical aspects of data collection.

The paper is composed of five sections. Section 2 outlines the motivations for data collection and analysis. Section 3 defines the kind of reliability-related data to be collected during development and in operation and the kind of processing that can be performed. Section 4 discusses the relevance of software reliability evaluations and gives recommendations for setting up a data collection and analysis program. Section 5 summarizes the paper.

2. Objectives and motivations

The objectives of software reliability analysis can be generally placed in one of two categories: a single system (short-term objectives) or generations of multiple systems (long term objectives). Short-term objectives support the development and management of the present system while long-term objectives deal with planning future systems and the infrastructure needed to develop them. Typically, short-term objectives require a reduced amount of data to be collected on a limited period of time while long-term objectives require that more data be collected over a long period of time on several products.

Short-term objectives are achieved through timely and cost-effective feedback. These objectives, in turn, depend on the considered perspectives (i. e., the supplier or the customer perspective) and the considered lifecycle phase (development or operation and maintenance). When the software system is under development, the supplier interests (among other things) are to:

- Understand project progress towards completion. Specifically plan and execute the efforts needed to effectively and efficiently remove faults present in the program.

- Manage the development so as to plan exit from one phase to the following one.

- Ensure that the reliability meets user requirements, when explicitly specified or not.

A commonly stated aim is to develop a software system with "zero defects". As it is out of the scope of any analysis method, in practice, this aim should be changed to: developing a software system with "zero failure" or with a low failure rate. Which means that the software can have some faults (also referred to as defects) that will not be activated, or seldom activated, under the operational profile. The latter is a more achievable aim.

When the software is in operation:

- The supplier is interested in:

  - The maintenance effort needed, through the estimation of the expected number of failures among all installations (or the number of corrections to be performed).

  - Reduction of the time to resolve problems.

- The customer is interested in the impacts of the failures on their operation, i. e., the mean time to failure or failure rate (either the instantaneous failure rate or the expected residual failure rate in operational life).

The above objectives are not independent. Note that each objective may be composed of several sub-objectives; for example, considering the objective "Evaluate reliability ", one will usually be interested in the failure rates of the components, which in turn can be used to project impacts to the maintenance efforts and impacts to customers operations.

When the data are collected, immediate feedback is expected. However, it may be some time before all benefits are felt. In that case feedback may only be profitable to the subsequent generations of the product or to broadly similar products (i.e., long-term results).

Long-term objectives are usually reached through the observation of several generations of software systems and the accumulation of systematic knowledge about the systems themselves and the development process. Analyses of the software behavior, based on the observed failure data, allow identification of possible weakness. Once identified, these weaknesses can be adduced. Reliability analyses help the manager to anticipate the software behavior. Using results drawn to past experience, (s)he can plan more efficiently the development process. In this context, a program of software reliability improvement is an element of a more general software process maturity effort.

In addition to specific experiences, several papers and books have already been published advocating and defining methods for improving software process based, among other things, on data collection (see e.g., [Humphrey 1989; Musa 1998]).

One of the most common objections to software reliability programs is their cost. The relationship between the level of dependability required and the associated cost is a complicated one as it includes such factors as the supplier's rework cost, the maintenance cost and the failure consequences to the user. However, past experience has shown the cost of fixing a fault uncovered during operation to be at least one order of magnitude higher than the cost of the same fault detected during development [Boehm 1981]. When the costs to the user and the negative impacts to the producer reputation are included the effect is magnified.

The benefits from a reliability improvement program sometimes cannot be perceived in a single product lifecycle. In that case, the cost of a reliability program has to be regarded as an

investment for subsequent systems rather than as an overhead for a single system. Usually the gains are substantial — even if they are not always immediately felt. It is worth noting that all the companies that have followed a well-defined program for improving software process and quality agree on the fact that the benefits are worthwhile. However, it is very difficult to partition the gains according to the methods used (e.g., the relative impact of fault prevention and fault removal techniques is very difficult to be assessed).

The examples presented below indicate some observed benefits. They have to be considered within the context in which they have been obtained.

- The results obtained through the quality program started at AT&T's International DEFINITY PBX [Donnelly *et al.* 1992] (using among other techniques the Cleanroom approach and software reliability objectives) show:

  - A reduction factor of 10 in customer-reported problems.

  - A reduction factor of 10 in maintenance cost.

  - A reduction factor of 2 in the test interval.

  - 30% reduction in new product introduction interval.

- The IBM experience reported in [Ehrlich *et al.* 1993], shows that the benefit-to-cost ratio brought by such analyses is 6.14, 11.98, and 78.65 depending on whether the cost of a failure is 500, 5,000, or 50,000 monetary units. It was based on the analysis of reliability-related data (trend tests and reliability growth models) and the application of an economic model to determine optimal release time.

- The experience reported by Fujitsu [Aoyama 1993], using the concurrent development approach, shows that the release cycle has been reduced by 75%.

- Motorola (Government Electronics Div.) [Diaz and Sligo 1997] has followed an improvement and an evaluation program: they went from CMM level 2 to level 3 in three years and it took them three more years to reach level 4 for the whole process and level 5 for policies and procedures. From level 2 to 5, the number of faults has been divided by 7, the cycle time by 2.4 whereas productivity has been improved by 2.8.

- Bull NH Information Systems [Weller 2000] has applied statistical process control to two projects, based on analysis of data collected during inspection, unit test, integration and system test. For the two products, only one fault has been found in operation in a limited customer use, and the release fault density was more than 10 times better than previous releases and the test turns have been halved.

It can be argued that the above reported examples concern large and well-established companies working most of the time on large software projects. This is true. However, more recently published experiences show that following well-organized measurement programs is also worthwhile for small organizations (see e. g., [Kautz 1999] or [Grable *et al.* 1999]). Nevertheless, the data collection program should be adjusted to small companies to suit the company products, goals and means.

It is worth noting that reliability measurement leads the development team to discuss what to measure, how to measure it and how to utilize and capitalize upon the results. It favors communication among the team members. Moreover, the analyses preformed by the team serve to unite them and to make them aware of possible problems along with their abilities.

3. Data collection, validation and processing

The nature of data to be collected is strongly related to the objectives desired. This section will concentrate on data to be collected for **software reliability analysis**. We first introduce

examples of data items to be collected, then discuss the practical aspects of data collection and feedback, before addressing data validation and processing.

3.1. Data to be collected

A complete and detailed data collection program assists in understanding the phenomena involved but the effort required may be daunting to the staff performing the work. A balance between the amount of data to be collected and the subsequent effort has to be found.

In order to perform reliability analysis, the following types of data should be collected:

- Data characterizing the software: size, language, functions, verification and validation methods and tools, etc., this is being referred to as background information in [Grady 1992].

- Data describing failures and corrections: time of failure occurrence, nature and impact of the failures, current version of the software in which the fault has been activated, conditions under which the fault has been activated (the so-called operational profile), fault location, type of faults, versions in which the faults have been fixed, etc.

Background information is recorded at the beginning of data collection and when changes occur, such as the introduction of new functionality or new development methods. On the other hand, data related to failures and corrections are recorded each time a failure occurs or is fixed. Usually they are collected through use of *failure* and *correction reports* featuring well-defined headings that have to be filled in either by the individual who observed or documented the failure or by the persons in charge of handling modifications. The reports must be constructed in such a way that the information provided is well structured, pertinent and easy to fill in. Ideally, a large part of it should be composed of short tick-off questions. In many cases, data is recorded using electronic forms. Good data collection procedures can be found in [Basili and Weiss 1984].

Figure 1 summarizes at a high level the types of information that are required to initiate a reliability study and lists desirable information that helps in performing a more detailed study.

Failure Report (FR)	Correction Report (CR)
Required Information 　Serial number (report identification) * 　Reporting individual 　Product reference, version affected 　Reporting installation 　Date and Time of failure occurrence **Desirable Information** 　Failure occurrence conditions 　Failure severity (or consequence) 　Affected function (or task) 　Action proposed (if any)	**Required Information** 　Serial number (report identification) * 　Reporting individual 　Product reference, version(s) modified 　Date and Time of correction 　Correction description **Desirable Information** 　Modified component(s)

* Required to make correlation between failures and corrections.

Figure 1. Information to be collected.

We recommend collecting both required and desirable information. By way of example, the desirable information can be used as follows:

- Consequences of failures: to follow up software reliability with respect to certain failure modes or some critical tasks, e.g., one can only consider the most severe consequences for safety evaluation.

- Fault(s) that have been corrected and the location of faults: to evaluate the reliability of the components; this is very useful when components are re-used for other applications (which is of prime importance in industry).

- Conditions under which failures occur: to enable reproduction of failures for fault diagnosis, and to investigate the workload influence on software behavior.

It may be impractical to create a failure and correction report for each incident during the early development phases, due to the large number of failures that may occur. Hence, data may

9

first be collected in the form of "number of failures" per day or week, according to the number of failures observed and the duration of the activity. The information derived from such data collection and analysis is limited in scope but may be useful for these early phases. For example, recording only the number of corrected faults per component allows evaluation of the fault density of the components, but does not allow its reliability to be directly evaluated. Later, during final system testing, initial field testing, deployment and operation, the unit of time should be expressed in terms of execution or operation time in an operational environment.

The specific data items to be collected result from an agreement between the engineers and the manager(s) who should also agree on common goals from the beginning. It is worth noting that data collection is an iterative process: most of the time, after a first period of data collection and early analyses, the data collection reports are modified (either improved or refined) to adapt to the specific situation.

As manual data collection can be cumbersome, an automated collection process is often used. The main drawback of an automated process is that the project may not take absolute time to define a sufficient and yet minimal set of data items. As it is easy to collect data automatically, projects often attempt to collect as many data items as possible to insure they have sufficient information to support later analysis. When the analysis is performed, the project finds that it has so much data that it is unable to process everything. Our recommendation is, if data are to be collected automatically, the goals should be precisely defined and the data needed to support the decision-making process should be well understood from the beginning. This should be done in the same manner as if it is to be performed manually (paper-based or entered in an electronic form).

3.2. Data collection organization

Measures to be collected have to be defined unambiguously according to the goals of the study. Also, the role assigned to each staff member involved in collecting data should be clearly defined. This includes the project manager, the development, the validation team and the users. Since motivation and training are key factors, all personal should be briefed about the objectives of data collection and quick feedback (even if initially incomplete, due to lack of information) should be provided to motivate all relevant personal. Likewise, it should be made clear to the entire staff that the data will only be used to manage and follow up the product and the production process and will not be used to assess their individual performance. This is mandatory to creating a good working atmosphere and establishing mutual confidence in the process.

Indeed, the staff will be more readily accept the effort appended during data collection if they are convinced of its necessity and benefits, either immediately or later. In addition, timely resolution of problems detected early can prevent occurrences of further problems later.

Data collection should be included in the development process and not be considered on its own. For example, if other data collection procedures are already in place for other objectives (such as for configuration management), additional information might be added to an existing report. Though the collection of software reliability data is most beneficial to future software products (long-term objectives), feedback is the best way to motivate the participants in the data collection effort (short-term objectives). The participants should be involved in the process of explaining trends identified by later data analysis, as well as in finding solutions to adverse trend. A timely return on the effort of collection will encourage the personal involved. They will feel more responsible and more concerned about data collection if they will benefit from immediate analysis. Even if they contribute to the improvement of subsequent projects, they also contribute

(may be to a less extent) to the improvement of their ongoing project. This is usually based on the perceived quality of the collected data. Hence validation is critical.

3.3. Data validation

Very often, the collected data cannot be used in a raw form. It must be checked for consistency and completeness. Validation is thus needed before reliability analysis. Data validation constitutes a fundamental step: its aim is to keep only genuine software fault data in the reliability database. It consists in a) filtering of the data to discard reports that are not relevant to the reliability improvement effort to software and b) identification of outliers.

Data filtering Usually the collected data includes reports documenting actual software failures or errors detected during software inspection but may also include extraneous data such as: false trouble reports (e.g., inconsistent data or hardware failures) or duplicated data (i.e., multiple reporting of the same failure). Filtering of the data is therefore needed to retain only genuine and relevant software faults, i.e., faults activated during software execution (software failure data) and faults detected during design or code inspection.

Careful filtering can be time-consuming and cumbersome. It necessitates precise knowledge about the system and the software as well as interviews of the persons involved in testing and data collection. When the failure reports are in an electronic form, a part of the filtering activity can be automated.

Even if the reasons for discarding a part of the collected data items may vary from one system to the other, the retained FRs may constitute only 50 to 60% of those recorded. This phenomenon has been reported in [Basili and Weiss 1984; Laryd 1994; Levendel 1995] and [Kaâniche *et al.* 1990; Kaâniche and Kanoun 1996]. For instance, among the 2146 FRs analyzed in [Kaâniche *et al.* 1990], only 1172 (55%) have been identified as corresponding to software

12

faults. The partition of the other fault origins is given in Figure 2-a, with duplicated failure reports constituting a large proportion of the discarded reports. On the other hand, the 3063 FRs analyzed in [Kaâniche and Kanoun 1996] produced 1853 FRs that corresponded to true software faults (i. e., 61 %). As indicated in Figure 2-b, 24% were unusable and the other 15% correspond to documentation, hardware or "Others". Thus filtering allows the identification of weakness in the data collection process.

It is worth noting that multiple manifestations of the same fault (usually called rediscoveries) have to be kept since they correspond to unique failure instances (occurring on the same or different installation) and thus they affect the observed reliability.
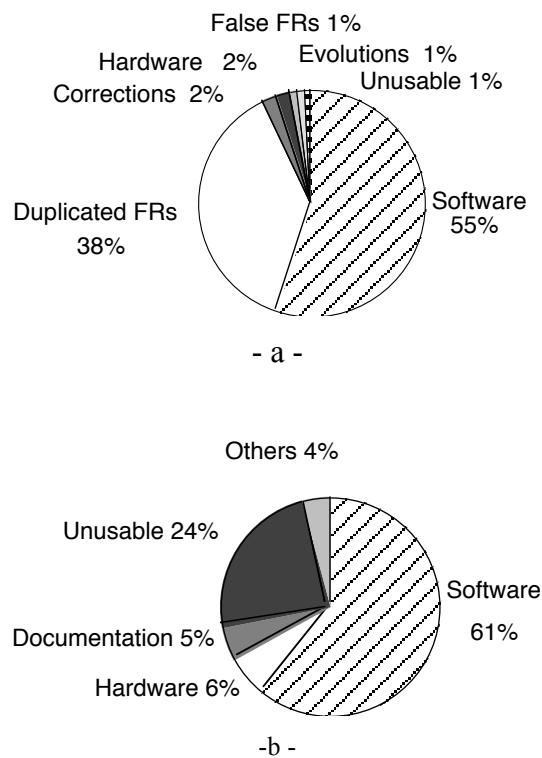

Figure 2. Examples of results from data validation.

Outliers. An observation that is atypical of the majority of observations is considered an outlier. Outliers may correspond to abnormal operational conditions or may indicate subtle interactions within the software itself. A number of statistical tests have been proposed (see e. g.,

13

[Aïvazian 1970]) to identify outliers (that are usually detectable among the highest and the lowest values). Including outliers in the analysis may significantly change the conclusion. On the other hand, excluding them may lead to misleading conclusions if in fact they correspond to correct observations.

In the context of software reliability growth, an outlier corresponds to a high or a low value of the time between failures:

- High values of outliers occurring at the end of the observation period are generally due to a meaningful reliability growth. Such outliers should be retained.

- Low values at the beginning of this period usually correspond to a low initial reliability and also should be kept.

- High and low isolated values in the middle of the period, usually, do not correspond to (and are not representative of) the "normal" behavior of the software and as such they can be eliminated from the data set. However, such occurrences may indicate unusual usage profile.

In one of the systems that we have studied [Kanoun *et al.* 1997], we were faced with the following situation. The average number of failures at the beginning of the testing period was "a few failures per week", then no failures during 45 days, then again, a few failures per week. All of them origin from genuine software faults. This corresponds to an initial failure interval of the order of some hours or days, then a failure interval of 45 days then failure intervals of some hours or days. After discussion with the developers, we learned that indeed the test of this software system was stopped during this intermediate period and the testing team was re-assigned to another software system.

In practice, before taking any decision, it is best to consult with the personal involved in the development process to determine if the data identified as outliers are representative of software behavior. An alternative is to perform the analyses with and without the outliers and then compare the results to determine the influence of such data items.

Retained data set. Filtering — and removal of outliers when necessary — results in a database of FRs and CRs corresponding to software faults activated under "representative" conditions. This step is essential as the analysis is carried out on that retained data set.

From FRs and CRs, the *times between failures*, the *failure intensity* or the *failure rate* can be computed and retained ideally in electronic files. Selecting which metric to use should be guided by the objective of the reliability study and the lifecycle phase. Using "failure intensity" (i. e., the number of failures per unit of time) will reduce the impact of local fluctuations in the data. The unit of time used for computing the failure intensity is a function of the type of system usage, the number of failures occurring during the considered units of time and the duration life-cycle phase considered.

3.4. Data processing

Processing the retained data set consists mainly of performing statistical operations. This consists of three activities: descriptive analyses, trend analysis and reliability evaluation. They are merely outlined in this section, as more details can be fond in our previous published papers (see e.g., [Kanoun and Laprie 1996; Kanoun *et al.* 1997]). In order to make detailed analyses, the collected data set should be partitioned into subsets.

Data partitioning into sub-sets is needed whenever a specific detailed analysis is required. The most common basis for such partitions are failure severity (i. e., consequences) and fault location (i.e., faulty components). Other partitions may also be used including failure occurrence

conditions, the nature of faults or the type of hardware installation. The latter partition is used when the software is in operation with several copies installed on different installations.

Applying reliability growth models to the most severe failures allows us to evaluate the software failure rate corresponding to the most critical failures. This failure rate is generally more significant than the overall software failure rate which incorporates failures that may not have major impact on system operation.

By way of example, the global failure rate of the software switching system analyzed in [Kanoun and Sabourin 1987] is $3.8 \; 10^{-6}$/h whereas the rate of failures having the most critical consequences is only $1.2 \; 10^{-7}$/h (which is almost 30 times lower).

Correspondingly, the evaluation of failure rates of the various components of a system will determine the influence of each component on the whole failure rate and serves to favor corrective actions on the ones of lowest reliability.

To conclude, data validation and partitioning activities should be specific to the system. They depend on the way data is collected and recorded, the purpose and the structure of the software and the objectives of the analysis. All these activities may not be needed for every reliability study. For instance, if the data items are entered in a database and checked as they are entered, validation is not needed. Data partitioning is performed or not depending on the level of detail needed. We have attempted to provoke some general and practical recommendations above.

Descriptive analysis consists in synthesizing the observed phenomena to identify the most underlying ones. It may consist of simple analyses such as:

- The distribution of faults among software components (such as new, modified or re-used).
- The fault typology.

- The fault density.

 We may also perform combined analysis such as the relationship between:

- The conditions of failure occurrence and failure severity.

- The fault location and failure severity.

- The fault density of the components and their size.

 Other interesting statistics not necessarily directly related to reliability can be used to give valuable information on the efficiency of the correction and maintenance process:

- Number of components modified to correct a fault. This evaluates the independence of components.

- The number of incomplete or bad corrections, and fault reintroduction factor (regression faults). This gives valuable information on the software process.

- The number of rediscoveries. This gives insight into the expected frequency of faults in the deployed system.

- The link between test coverage and observed fault density. Again, this provides insight into the testing process and the quality of the product.

- Time delay between failure occurrence and fault resolution.

 By way of example, for the system studied in [Kaâniche and Kanoun 1996], Figure 3 shows that while two thirds of faults are removed within six months, 22% of faults are removed 7 to 12 months after their detection and 12 % of faults are not removed one year after their detection. This time delay measures the responsiveness of the organization and it is also related to the complexity of maintaining the product. The analysis of the reasons of large delays may help improving the maintenance process.
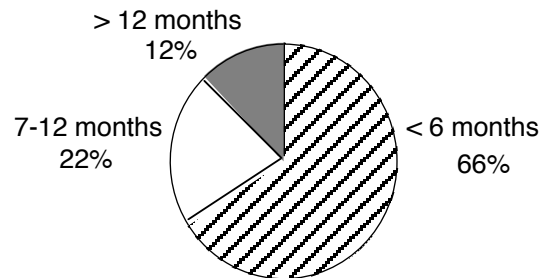
Figure 3. Time delay between failure occurrence and fault resolution.

Descriptive analysis is very useful and is used by many companies [Grady and Caswell 1987; Ross 1989]. Accumulation and analysis of information about several projects, products and releases allows organizations to a) have better insights into their products and b) establish links between their products and the development and maintenance process.

Reliability trend analysis. Removing faults, the underlying causes of failures, result in reliability improvement or growth. However, this is not always the case, as reliability may also decrease. This does not necessarily mean that the software has more faults; it is only an indication that the new usage conditions produce more failures per unit of time. For example, the earlier failures may have inhibited the users from exercising portions of the product. Correcting the faults that produced the early failures may now expose other faults.

Decreases in reliability result from a variety of reasons including:

• Variation in the utilization environment: variation in the testing effort during debugging, change in test sets or test environment, addition of new users during the operational life and so on.

• Dependence between faults: some faults may be masked by other faults and they cannot be activated as long as the masking faults are not removed.

• Addition of new features to an existing system: This changes both the operational profile and the product itself.

18

- The introduction of new faults when correcting early faults. Depending on the magnitude of the correction and its input (such as described above) this is more or less likely.

A reliability decrease at the beginning of a new activity (such as a new life cycle phase, a change in the test sets within the same phase, the addition of new users or activation of the system in a different profile of use) is generally expected and considered normal. Trend analysis allows this kind of behavior to be explicitly observed. A long lasting decrease may indicate fundamental design or process problems. Analyzing the reasons for such persistent decrease as well as the nature of the faults is critical.

Trend analyses are used in evaluating the efficiency of test activities and in controlling their progress. The timely identification of decreasing reliability allows the project manager to make the appropriate decisions needed to avoid problems. It can help identify what parts of the software to re-examine. In general, the role of trend analyses is to draw attention to problems that might otherwise pass unnoticed until it is too late, thus providing an early warning. Trend analysis is used in many companies (see e.g.[Grady and Caswell 1987; Ross 1989]). Several statistical tests can be used to identify local and global trend such as Laplace or subadditivity tests (see e.g., [Kanoun and Laprie 1996; Kanoun *et al.* 1997]).

Reliability evaluation. A software reliability growth model is a formal representation of a random process through which software reliability (or a directly related measure such as the failure rate, the failure intensity) is characterized and predicted as an explicit function of a variable which is the number of failures or the time. More than fifty reliability growth models have been developed during the last decades, and several publications have been devoted to these models (see for example [Musa *et al.* 1987] or [Xie 1991]). Earlier reliability growth models for hardware have been used for decades.

Reliability growth models may lead to non-realistic results when the trend displayed by the data differs markedly from that assumed by the model. However, when applied to data displaying trends consistent with their assumptions, predictions can be quite useful as shown in our previous publications [Kanoun and Sabourin 1987; Kanoun *et al.* 1991; Kanoun *et al.* 1997]. The already existing reliability growth models only allow two types of behavior to be modeled: a) decreasing failure intensity (reliability growth) or b) increasing failure intensity prior to undergoing decreasing failure intensity (S-Shaped models). Using trend analysis, failure data can be partitioned according to their trend and reliability growth models can be selected according to those trends. Tools such as SoRel [Kanoun *et al.* 1993] can be used to analyze the trend and apply appropriate reliability growth models.

In practice the models can be applied as long as the environmental conditions remain significantly unchanged (no major changes in the testing strategy or in the specifications, no changes in the operational profile and so on). If a significant change in the development or operational conditions takes place, great care is needed since changes in reliability trend may result rendering the assumptions or the models invalid.

4. Results relevance and recommendations

As a matter of practice, based on our experience, the control of the development activities as they occur allows timely detection of possible deviations. Corrective actions can thus be undertaken early to avoid increase of these deviations. We recommend the following:

- **Collect and validate** the data as early as possible during the development, starting from design and code inspection (i. e., static verification). Ideally this should be done in the broader context of process management and improvement.

- **Analyze** the retained data sets as soon as the validated data is available, to detect whether development has diverged from expected behavior. If this happens, diagnose the cause of this divergence and select the actions needed to counter it.

- **Provide feedback** to the collectors on a regular and timely basis, and solicit their participation to the analysis.

- As the development nears completion, use the data collected to **evaluate the current reliability** and apply reliability models, in preparation for deployment.

- At this stage, data from previous products are needed to **predict the reliability** that will be attained in operation. This may be done empirically by extrapolation, or by using a more formal approach such as the product-in process approach presented in [Laprie 1992].

- **Document and archive** the results in a manner that is easy to be exploited for the succeeding generations.
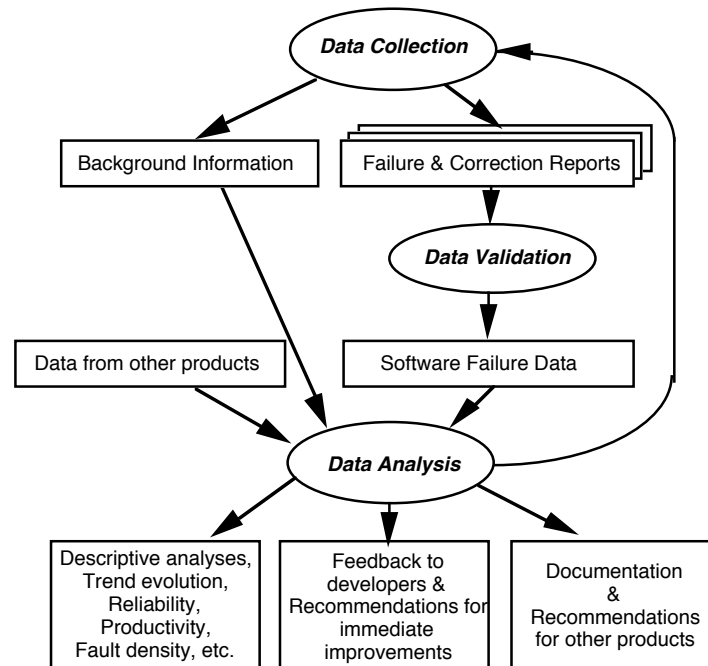
This is summarized in Figure 4.

Figure 4. Framework for software reliability analysis and improvement.

21

The relevance of the predictions obtained from reliability growth model and trend analysis varies with the life-cycle phase.

Thus:

- Application of growth models during the early stages of the development is usually not appropriate. The product and its environment are both evolving rapidly, negating the assumptions of the models. It is thus not very helpful and, in this case, reliability prediction should be guided by trend analyses as discussed earlier.

- When the program under evaluation becomes more stable, the times to failure should be larger and the results of reliability growth models are more convincing, particularly when the software is activated under its real operational profile or simulations.

- When the software is in operation on multiple installations, the results are usually of high relevance since we have true statistical failure data (see e.g., [Adams 1984]). Indeed, the practical current limit in the assessment of the failure rates, before operational use, is in the range of 10^{-2}/h to 10^{-4}/h. On the other hand, observed failure rates in operation range from 10^{-3}/h [Chillarege et $al.$ 1995] to a few 10^{-6}/h [Kanoun and Sabourin 1987; Gray 1990] or even down to $5 \cdot 10^{-8}$/h [Laryd 1994].

We followed the above steps in evaluating the reliability of multiple real software systems. Their characteristics are presented in Table 1. During the software validation phase, the main results concerned the study of reliability trend in response to debugging activities and the prediction of the number of faults that will be activated over the subsequent phases [Kaâniche et $al.$ 1990]. During operation, the objectives of reliability analysis were twofold: namely, the estimation of the maintenance effort and the failure rate of the software as perceived by an average user.

22

Some examples of the results obtained for these systems (which are electronic switching systems):

- For the E10-B, the evaluation of the residual failure rate in operation derived from the application of reliability growth models to failure data collected on the software [Kanoun and Sabourin 1987] allowed the system reliability of the ESS to be evaluated (both hardware and software).

System	Language	Volume	Duration+	Phase	# Systems	# FR and/or CR
E10-B	Assembler	100 k-bytes	3 years	Val./Op.	1400	58 FR / 136 CR
TROPICO-R 1500	Assembler	300 k-bytes	2 years 3 months	Val./Op.	15	465 FR/CR
TROPICO-R 4096	Assembler	350 k-bytes	2years 8 months	Val./Op.	42	210 FR/CR
TROPICO-RS	Assembler	420 k-bytes	3 years 11months	Op.	37	212 (+ 105*) FR/CR
TROPICO-RA	CHILL	815 k-LOC	5 years 8 months	Op.	146	3063 FR/CR
Telecom. Equip.	PLM-86	500,000 inst.	16 months	Val.	4	2150 FR
Work station	various	--	4 years	Op.	1	414 FR

+ Duration of the data collection period
Val.: Validation   Op.: Operation   FR: Failure Report   CR: Correction Report
inst.: instructions   k-LOC: kilo Lines Of Code   *: detected by code inspection

Table 1. Software systems whose reliability has been analyzed at LAAS.

- For the TROPICO-R 1500, 4096 and RA we followed two complementary approaches [Kanoun *et al.* 1991; Kaâniche and Kanoun 1996; Kanoun *et al.* 1997]:

  - From the supplier's point of view: estimation of the maintenance effort needed in operation to address the failure reports (FRs) issued by the customers.

  - From the customer's viewpoint: estimation of the residual failure rate in operation to evaluate the impact of software reliability on the ESS system reliability.

23

- In the three first TROPICO ESS generations (1500, 4096 and RS), the comparative analysis performed in [Kaâniche *et al.* 1998] focused on fault density of the three products and their components. The results gave insight into the influence of software development environment on the reliability evolution of the family of products. The estimated failure rates for the TROPICO ESS generations are in accordance with those observed after one year of operation: they lie in the range of 10^{-5}/h to 10^{-4}/h for the whole software system including all failures, irrespective their consequences.

5. Conclusions

Reliability measurement is the first step of a software reliability program within a company. However, establishing a data-collection process for reliability analysis entails overhead cost, which may be an obstacle mainly during development. However, the experiences reported in Section 2 show that the improvements obtained more than offset these costs. In fact, they show that the gain in productivity and reliability is obtained at almost no extra cost, and even with an overall cost reduction for most of them.

Section 3 presented some measures that can be gathered during development to help improve the reliability of the product. As measurement is itself based on a data collection process the collected data can contain errors. The raw data set has thus to be validated and the retained data set is then processed to obtain useful information on reliability. The results help in managing the software reliability during development. We put emphasis on short-term objectives, implying timely and efficient feedback for the ongoing project and showed the kind of results that can be expected from data collection and processing.

Acknowledgement

References

Adams, N. (1984), "Optimizing Preventive Service of Software Products", *IBM Journal of Research and Development* 28 (1), 2-14.

Aïvazian, S. (1970), *Statistical Study of Dependencies*, Editions Mir, Moscou, *in French*.

Aoyama, M. (1993), "Concurrent-Development Process Model", *IEEE Software,* July, 46-55.

Basili, V.R. and D.M. Weiss (1984), "A Methodology for Collecting Valid Software Engineering Data", *IEEE Transactions on Software Engineering* 10, 6, 728-738.

Boehm, B.W. (1981), *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ.

Castelli, L., B. Coan, J.P. Harbison and E.L. Miller (1997), "Tradeoffs when Integrating Multiple Software Components into a Highly Available Application", In *Proceedings of the 16th IEEE Symposium on Reliable distributed Systems*, IEEE Computer Society Press, Durham, NC, USA, pp. 121-128.

Chillarege, R., S. Biyani and J. Rosenthal (1995), "Measurement of Failure Rate in Widely Distributed Software", In *Proceedings of the 25th IEEE International Symposium on Fault Tolerant Computing (FTCS-25)*, Pasadena, Ca, pp. 424-433.

Diaz, M. and J. Sligo (1997), "How Process Improvement Helped Motorola", *IEEE Software* September, 75-81.

Donnelly, M.M., J.D. Musa, W.W. Everett and G. Wilson (1992), Best Current practice: Software Reliability Engineering, AT&T Bell Laboratories Software Quality and Productivity Cabinet.

Ehrlich, W., B. Prasanna, J. Stampfel and J. Wu (1993), "Determining the Cost of a Stop-Test Decision", *IEEE Software* March, 33-42.

Grable, R., J. Jernigan, C. Pogue and D. Divis (1999), "Metrics for Small Projects: Experiences at the SED", *IEEE Software* March/April, 21-29.

Grady, R.B. (1992), *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, Englewood Cliffs, NJ.

Grady, R.B. and D.L. Caswell (1987), *Software Metrics: Establishing a Company-Wide Program*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA.

Gray, J. (1990), "A Census of Tandem System Availability Between 1985 and 1990", *IEEE Transactions on Reliability* 39, 4, 409-418.

Humphrey, W.S. (1989), *Managing the Software Process*, Addison-Wesley.

Kaâniche, M. and K. Kanoun (1996), "Reliability of a Commercial Telecommunications System", *In Proceedings of the 7th International Symposium on Software Reliability Engineering*, White Plains, NY, USA, pp. 207-212.

Kaâniche, M. and K. Kanoun (1998), "Software Reliability Analysis of Three Successive Generations of a Switching System", *In Proceedings of the IEEE Workshop on Application-Specific Software Engineering and Technology (ASSET'98)*, IEEE Computer Society, Richardson, TX, pp. 122-127.

Kaâniche, M., K. Kanoun and S. Metge (1990), "Failure Analysis and Validation Monitoring of a Telecommunication Equipment Software System", *Annales des Telecommunications, In French* 45, 11-12, 657-670.

Kanoun, K., M. Bastos Martini and J. Moreira de Souza (1991), "A Method for Software Reliability Analysis and Prediction—Application to The TROPICO-R Switching System", *IEEE Transactions Software Engineering* SE-17, 4, 334-344.

Kanoun, K., M. Kaâniche and J.-C. Laprie (1997), "Qualitative and Quantitative Reliability Assessment", *IEEE Software* 14 (2), 77-87.

Kanoun, K., M. Kaâniche, J.-C. Laprie and S. Metge (1993), "SoRel: A Tool for Reliability Growth Analysis and Prediction from Statistical Failure Data", *In Proceedings of the 23rd International Symposium Fault-Tolerant Computing (FTCS-23)*, Toulouse, France, pp. 654-659.

Kanoun, K. and J.-C. Laprie (1996), "Trend Analysis", *Handbook of Software Reliability Engineering*, Ed. M. Lyu, Mc Graw Hill, Chapter 10, pp. 401-437.

Kanoun, K. and T. Sabourin (1987), "Software Dependability of a Telephone Switching System", *In Proceedings of the 17th IEEE International Symposium on Fault-Tolerant Computing (FTCS-17)*, Pittsburgh, PA, USA, pp. 236-241.

Kautz, K. (1999), "Making Sense of Measurement for Small organizations", *IEEE Software* March/April, 14-20.

Laprie, J.-C. (1992), "For a Product-in-a Process Approach to Software Reliability Evaluation", *In Proceedings of the 3rd International Symposium on Software Reliability Engineering*, Raleigh, NC, USA, pp. 134-139.

Laryd, A. (1994), "Operating Experience of Software in Programmable Equipment Used in ABB Atom Nuclear I&C Applications", *In Proceedings of the IAEA, Technical Committee Meeting*, Helsinki, Finland, pp. 1-12.

Levendel, Y. (1995), "The Cost Effectiveness of the Telecommunication Service Dependability", *Software Fault Tolerance*, Ed. M.R. Lyu, John Wiley & Sons, pp. 279-314 (Chapter 12).

Musa, J. (1998), *Software Reliability Engineering*, Computing McGraw-Hill.

Musa, J.D., A. Iannino and K. Okumoto (1987), *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, New-York.

Ross, N. (1989), "The Collection and Use of Data for Monitoring Software Projects", *Measurement for Software Control and Assurance*, Ed. B.A. Kitchenham and B. Littlewood, Elsevier Applied Science, London and New York, pp. 125-154.

Weller, E.F. (2000), "Practical Applications of Statistical Process Control", *IEEE Software* May/June, 48-55.

Xie, M. (1991), *Software Reliability Modeling*, World-Scientific, Singapore.